Lightweight HTTP Server for Embedded Systems

Epsilon HTTP

Andon M. Coleman

CNT 4104 – Computer Network Programming

Dr. Janusz Zalewski

May 02, 2011

Abbreviations and Acronyms

API	Application Programming Interface
BSD	Berkley Systems Distribution
C++	An Object Oriented programming language derived from C
HTTP	Hyper Text Transfer Protocol
HTML	Hyper Text Markup Language
IPv4	Internet Protocol – Version 4
OS	Operating System
POSIX	A set of standards for UNIX-like Operating Systems
RFC	Request For Comments
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
VFS	Virtual Filesystem
VxWorks	A commercial Real-Time Operating System by Wind River
eHTTP	Epsilon HTTP

1. Introduction

In the world of communication protocols that transfer human-readable content, none are more ubiquitous than HTTP. These days, humans have a wealth of Internet connected devices that understand HTTP and HTML at their disposal, from personal digital assistants to personal computers, to video game consoles, and even cell phones. Therefore, it makes a lot of sense to use the HTTP protocol to report remote system status to human operators.

The HTTP protocol uses connection-oriented Transmission Control Protocol sockets over the Internet Protocol, and serves requests by mapping URL paths to locally accessible resources. In general, this means that an HTTP server continuously listens on a specific TCP port for incoming connections, and uses request URLs to determine which file the client is interested in.

This generalized description breaks down in many specialized applications, and the difficulties associated with implementing the HTTP protocol in resource limited applications form the basis of this project.

2. Definition of the Problem

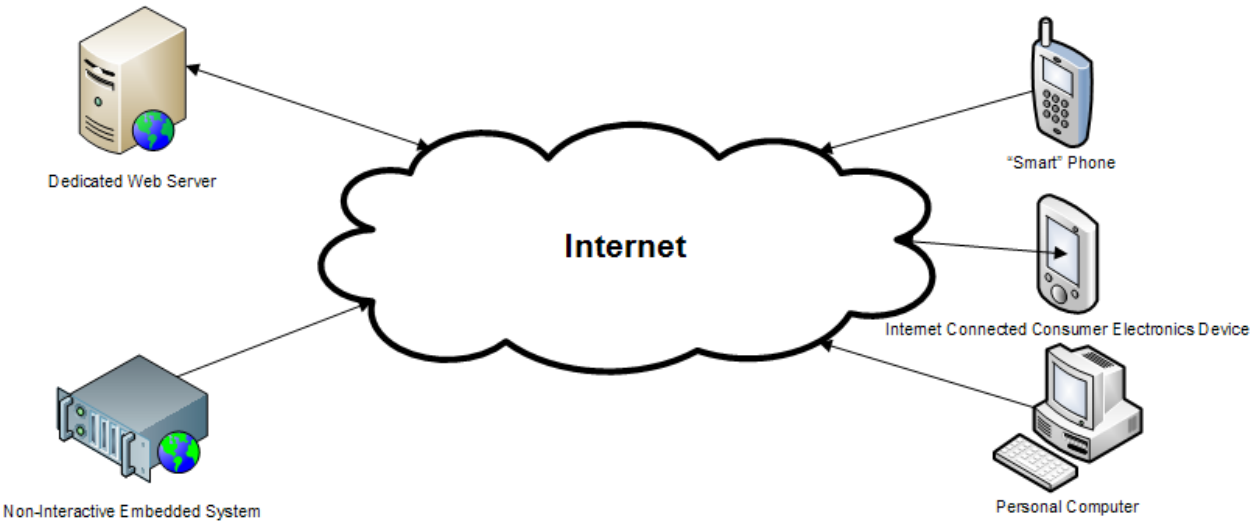


Figure 2.1 Physical Design Diagram

HTTP servers are usually large over-complicated suites designed to support every web technology under the sun. This makes using them in embedded applications impractical. Consequently, this project (Epsilon HTTP) aims to develop a light-weight HTTP server suitable for capability limited systems.

Lightweight, in this context, refers to the lack of dependence on third-party libraries, minimal implementation, and various other policies intended to reduce system requirements. The minimum system requirements for Epsilon HTTP are an IPv4 network stack, with support for BSD-style connection-oriented sockets, a C++ compiler and an Internet connection (Figure 2.1).

Since the server's target domain is embedded applications, fault tolerance is a critical part of the design. For instance, the server must be capable of dealing with malformed, or random input after accepting a TCP connection on its listening port. No attempt to parse messages that do not contain "HTTP" should be made, and the server should send a response to the originating host and immediately close the connection. Failure to effectively deal with malformed requests may leave the server inoperable and require human intervention to correct, which is not an easy task in embedded applications.

Epsilon HTTP is a completely new body of work, with the goal of developing a custom HTTP server capable of running the CGI programs used in the VxWorks Kernel Connectivity

project from previous semesters. [1] Its name is derived from the Computer Science concept of Machine Epsilon, which represents the smallest distinguishable difference in floating-point numbers; or in layman's terms, a "really small" number. Epsilon HTTP is by definition a "really small" HTTP server, so the name fits.

3. Proposed Solution

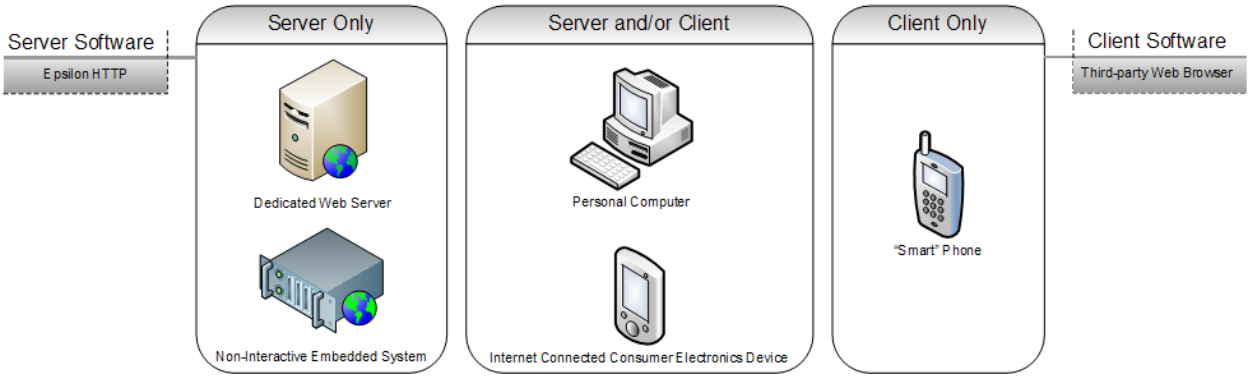


Figure 3.1 Application Domains

The most unique aspect of this project is the way that the HTTP server maps paths to files. Traditionally, a path such as /cgi-bin/foo.cgi refers to a location on an operating system managed filesystem. To accommodate VxWorks [2], this HTTP server will have to implement a Virtual File System, where directory structures and files have no operating system interaction. Reading and writing to, or executing a file in the Virtual File System reads or writes to a pool of process-managed memory, or calls a user-defined callback. The server may operate using a completely virtual file system, or using a traditional path to operating system file system (if the platform supports it).

Additionally, CGI may operate in the form of a blocking callback rather than forking the host process and running a binary executable. This allows the HTTP server to run rudimentary CGI on systems that do not support multi-tasking. It is an important design feature for VxWorks, as the HTTP server may be run from a boot loader before the VxWorks kernel is even loaded.

Another unique characteristic of Epsilon HTTP is the way it listens for HTTP connections. Traditionally, HTTP servers are implemented in the form of dedicated daemon processes that run as services in the background on a multi-tasking operating system. Epsilon HTTP's design supports non-multi-tasking operating systems, and it does so by constructing server instances within a host application's process. In this mode of operation, incoming connections are accepted and HTTP requests are processed when the host application determines it is appropriate to do so (usually as a stage within the application's main loop).

The HTTP/1.1 protocol defines at least 9 different types of requests, and 50 standard status codes. Dedicated general purpose HTTP servers such as Microsoft IIS [3] and Apache [4] are designed to meet or exceed all of these standard requests and status codes. However, an HTTP implementation is considered functional even if it only implements two request types (GET and HEAD), and three status codes (200 OK, 404 Not Found, 501 Unimplemented). Given the added requirement of serving CGI, Epsilon HTTP will only need to implement the HTTP GET, POST and HEAD requests.

Figure 3.1 identifies the participants in the client / server role when Epsilon HTTP is used. The software's design means it is capable of running on a wide variety of platforms. In fact, many devices are capable of both hosting, and connecting to an Epsilon HTTP server. However, "smart" phones are not as smart as they would like to think; platforms that do not support development in C++ are among the list of devices that Epsilon HTTP cannot fill the server role on.

4. Implementation

The organization of Epsilon HTTP focuses on a collection of connected components, presented in the order of operation. When porting Epsilon HTTP to a new platform, the socket (see section *4.1*) and file (see section *4.2*) backends should be the only components requiring modification.

4.1 Socket Backend

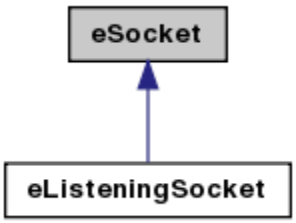


Figure 4.1 Inheritance Graph for eSocket

Epsilon HTTP requires support for TCP-based listening sockets. The default implementation wraps the BSD socket API in a thin layer in socket.h and socket_listening.h. As seen in Figure 4.1, eListeningSocket is an extension of eSocket, with added functionality necessary for connection-oriented socket communication (i.e. TCP).

4.2 File Backend

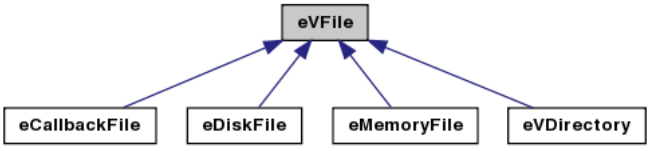


Figure 4.2 Inheritance Graph for eVFile

Epsilon HTTP supports two types of file systems, with a shared interface for ease of use. Figure 4.2 shows the described inheritance relationship.

4.2.1 Operating System Managed File System

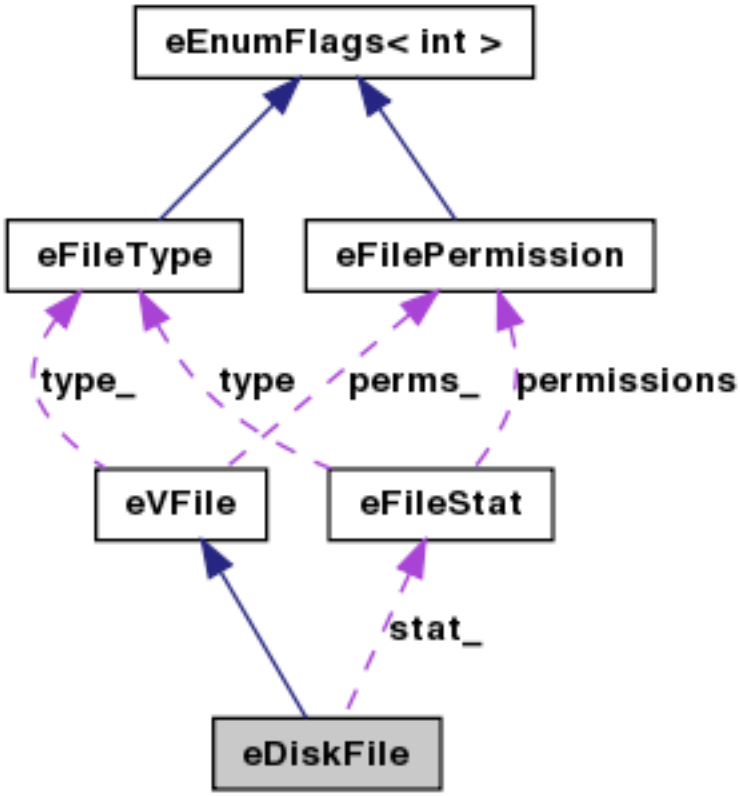


Figure 4.3 Collaboration Graph for eDiskFile

This mode uses a mix of POSIX file API and C stdlib file API routines to manipulate files. In addition to file I/O, the OS file system interface also provides an interface to execute a file and store its text output, as necessary for CGI support. Figure 4.3 shows the internal workings of the eDiskFile class, it is important to note that eDiskFile caches Operating System file stats, and thus appears to duplicate functionality of eVFile; this is not the case.

4.2.2 Virtual File System

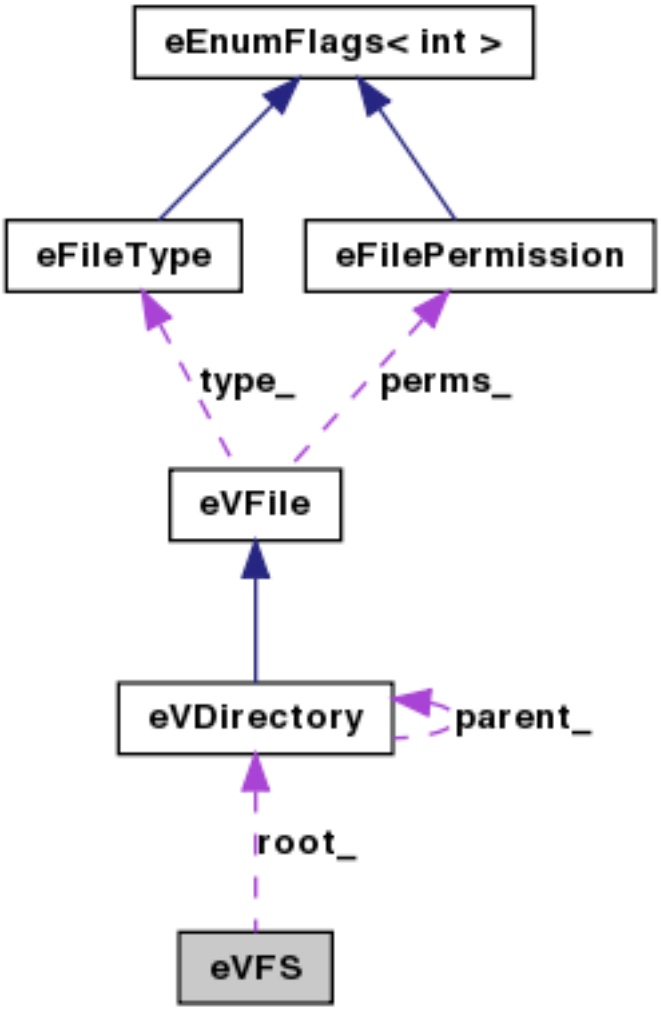


Figure 4.4 Collaboration Graph for eVFS

This mode forms a virtual hierarchy of files and directories, such that a file maps to memory addresses (*eMemoryFile* in *Figure 4.2*) or callback procedures (*eCallbackFile* in *Figure 4.2*) within the processes address space, rather than files on a disk. As seen in Figure 4.4, the basic structure of the VFS is based on a tree of Directories, with eVFile instances forming the leaf nodes.

4.3 HTTP/1.1 Support

Epsilon HTTP implements a small subset of the HTTP/1.1 protocol (better known as RFC 2616)

For more details on HTTP/1.1, see:

> “Hypertext Transfer Protocol – HTTP/1.1” [5]

The design is split between several specialized classes and subclasses, described below.

4.3.1 HTTP Client

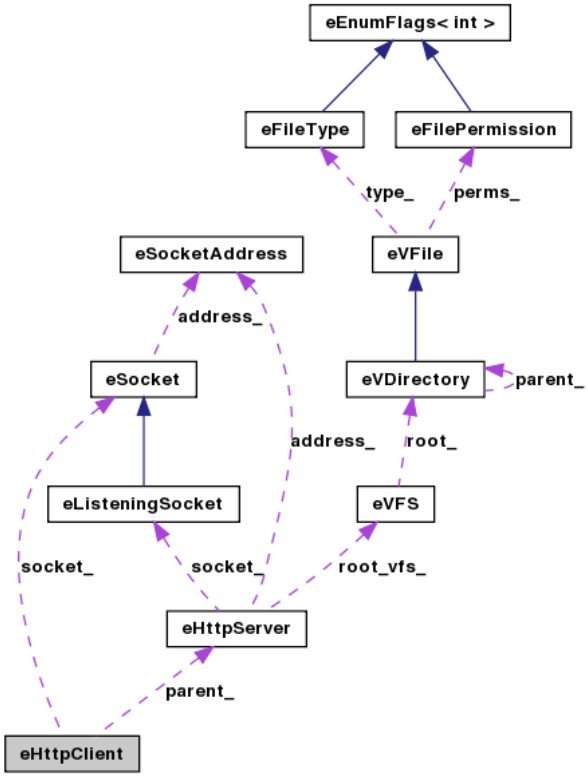


Figure 4.5 Collaboration Graph for eHttpClient

eHttpClient encapsulates an HTTP connection instance, it stores variables related to the connecting user agent (web browser), the server that the client is connected to, and provides the interface the server uses to relay HTTP responses (*4.3.4*). As Figure 4.5 shows, eHttpClient is an associative class; it associates a server instance with the socket instance used to communicate with the client.

List of Files for Directory: /

Parent Directory

DIR	CGI		Wed Mar 16 16:06:18 2011
DIR	Documentation		Tue Apr 5 10:49:49 2011
DIR	eDNS		Wed Mar 16 16:06:50 2011
DIR	eSMTP		Wed Mar 16 16:12:00 2011
DIR	eTextUI		Wed Mar 16 16:12:17 2011
4.83 KiB	file.cpp	r--	Sat Mar 5 23:51:09 2011
3.25 KiB	file.h	r--	Sat Mar 5 18:47:01 2011
3.21 KiB	file.o	r--	Sun Apr 3 22:23:41 2011
19.16 KiB	http.cpp	r--	Wed Mar 9 11:40:57 2011
11.29 KiB	http.h	r--	Mon Mar 7 23:01:21 2011
19.36 KiB	http.o	r--	Sun Apr 3 22:23:42 2011
34.38 KiB	httpd	r-x	Fri Mar 11 19:24:13 2011
1.18 KiB	httpd.cpp	r--	Fri Mar 11 19:23:54 2011
152 bytes	httpd.o	r--	Sun Apr 3 22:23:42 2011
72 bytes	mime.h	r--	Sat Feb 26 21:25:12 2011
24 bytes	robots.txt	r--	Mon Mar 7 13:50:40 2011
1.05 KiB	socket.cpp	r--	Sat Mar 5 19:22:58 2011
2.71 KiB	socket.h	r--	Fri Mar 11 00:33:28 2011
2.69 KiB	socket.o	r--	Sun Apr 3 22:23:42 2011
498 bytes	socket_listening.cpp	r--	Sat Feb 26 13:05:45 2011
307 bytes	socket_listening.h	r--	Sat Feb 26 13:10:38 2011
1.41 KiB	socket_listening.o	r--	Sun Apr 3 22:23:42 2011
540 bytes	vfs.h	r--	Sat Feb 26 13:38:32 2011

Epsilon HTTP 0.01a *(*):80*

Figure 4.6 HTTP Directory Index (as seen by client)

In Figure 4.6, a web browser has rendered the HTML output that eHttpServer (*4.3.2*) sent using the client interface – the automatic Directory Index feature is only available in OS file system mode (*4.2.1*).

4.3.2 HTTP Server

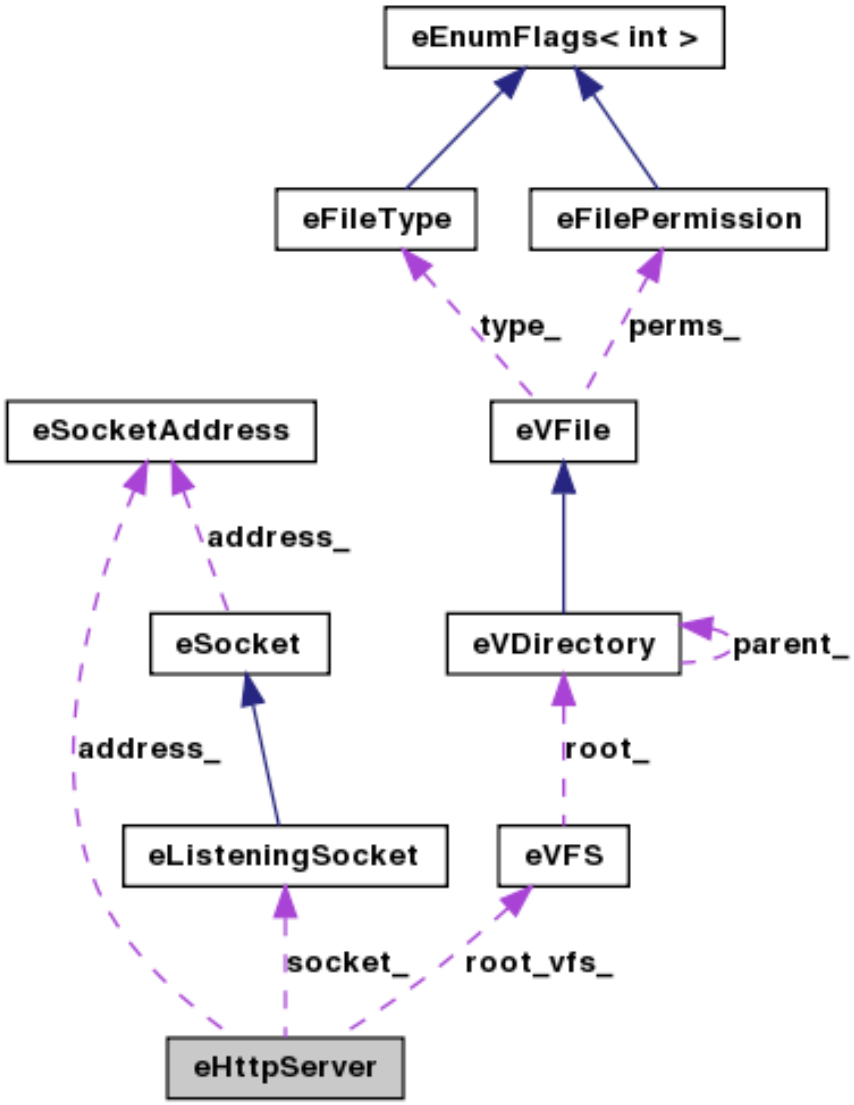


Figure 4.7 Collaboration Graph for eHttpServer

eHttpServer implements an HTTP server that listens on a dedicated user-defined TCP port. It is initialized with a root FS—virtual (*root_vfs_* in *Figure 4.7*) or OS-based—that it uses to map URL paths to files, and the TCP port it should listen on.

The server looks for incoming connection requests and processes pending HTTP requests whenever eHttpServer::think (…) is called. By “thinking” only when the host application can allocate run-time for this task, this allows the server to operate on systems that do not support multi-threading or multiple processes.

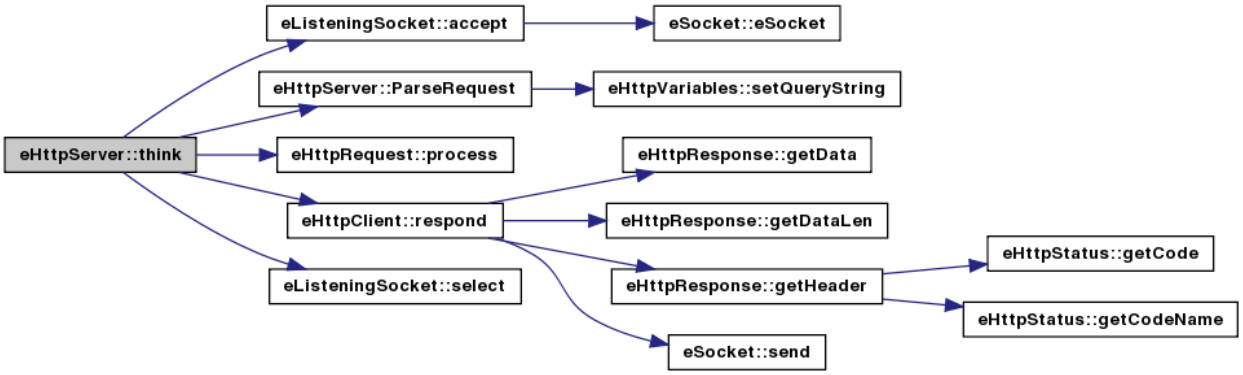


Figure 4.8 Callgraph for eHttpServer::think (…)

Figure 4.8 illustrates the primary operations preformed during a call to eHttpServer::think (…). The server uses eListeningSocket::select (…) to wait up to a user-defined limit of time for incoming connections. When the server receives a connection without timing out, it will parse the client request, process the request, and finally send a response to the client that created the connection. Otherwise, incoming connections are queued and will be handled during the next call to think (…).

Figure 4.9 HTTP Server Output (generated Directory Index in figure 4.6)

When the HTTP server receives a request, it parses the request type and constructs the appropriate eHttpRequest subclass to process the request (*4.3.3*). Figure 4.9 shows the result of processing and responding to an HTTP GET request.

4.3.3 HTTP Request

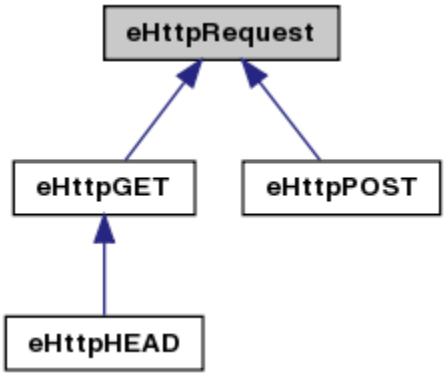


Figure 4.10 Inheritance Graph for eHttpRequest

eHttpRequest is a superclass for supported HTTP request types, it associates the client that generated the request with the server that serves it, and a specialized subclass of eHttpRequest (see *Figure 4.10*) implements the request's processing logic. In the process of processing a request, each of these subclasses will generate an HTTP response (*4.3.4*).

4.3.4 HTTP Response

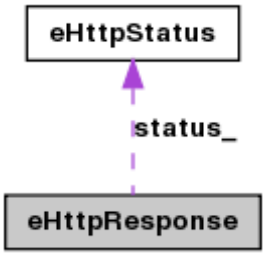


Figure 4.11 Collaboration Graph for eHttpResponse

eHttpResponse is returned when an HTTP request is processed, it contains an HTTP header string and (if the request type was not HTTP HEAD) 1 or more bytes of data, plus an HTTP status code (*status_* in *Figure 4.11*) that indicates the server's ability to service the request.

4.3.5 HTTP Status

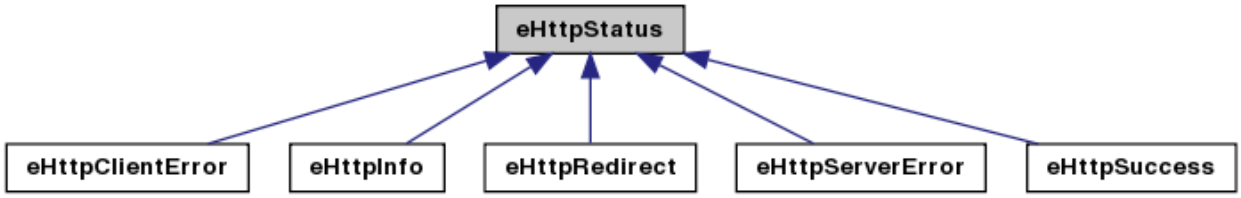


Figure 4.12 Inheritance Graph for eHttpStatus

Figure 4.12 illustrates the breakdown of status classes. Status codes are clustered into groups of up to 99 similar codes, in the format <Class>[Code]. Epsilon HTTP only implements Success (200-299), Client Error (400-499) and Server Error (500-599) status codes. The remaining status classes are reserved for future use.

4.4 CGI/1.1 Support

Epsilon HTTP implements a subset of the CGI/1.1 protocol (RFC 3875).

For more details on CGI/1.1, see:

"The Common Gateway Interface (CGI) Version 1.1" [6]

```
SERVER_SOFTWARE: Epsilon HTTP 0.01a
SERVER_PROTOCOL: HTTP/1.1


The local time is: Tue Apr  5 11:25:43 2011
```

Figure 4.13 Sample CGI Program Client Output

4.4.1 CGI Server

eCgiServer provides a common interface for executing CGI programs, each HTTP server instance is associated with its own eCgiServer.

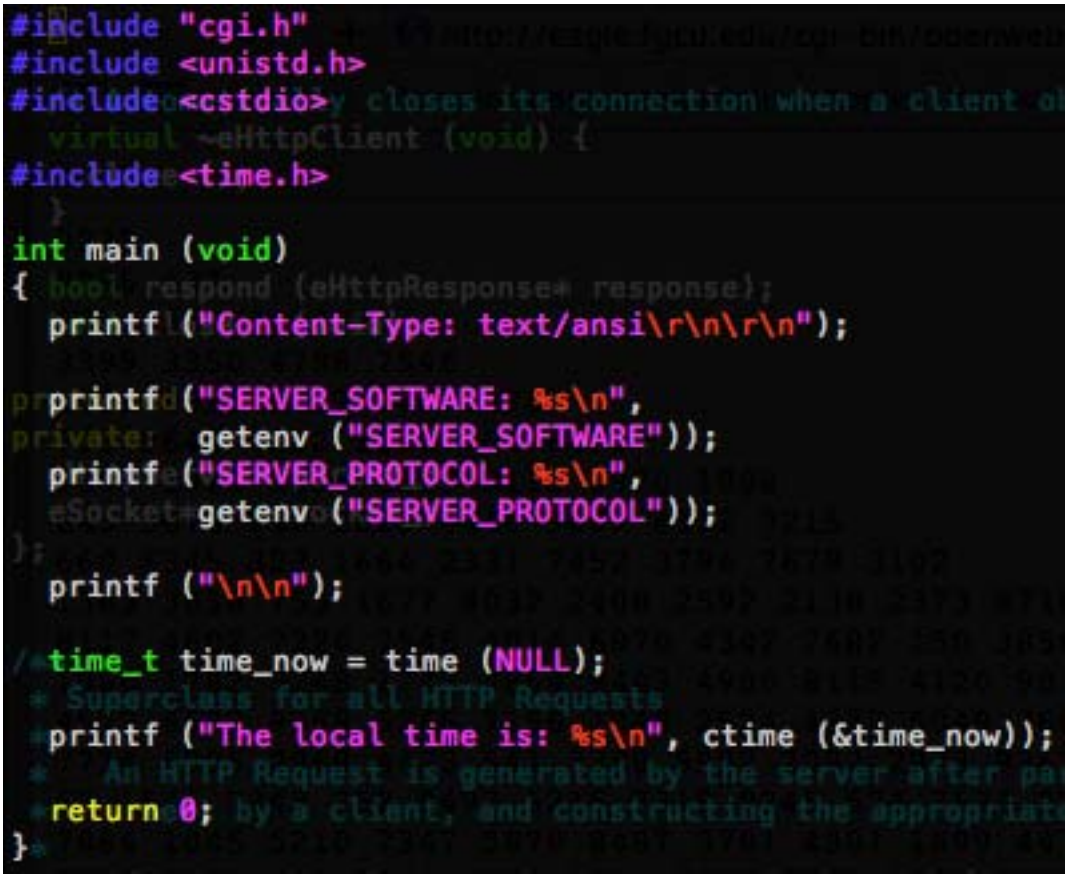
Figure 4.14 Sample CGI Program Server Output

In Figure 4.14, a client has requested a CGI program, and this has added an additional step (" >> Executing CGI …") to the process of servicing an HTTP request. The resulting client view of "/CGI/cgi_test" is shown in Figure 4.13.

4.4.2 HTTP Variables

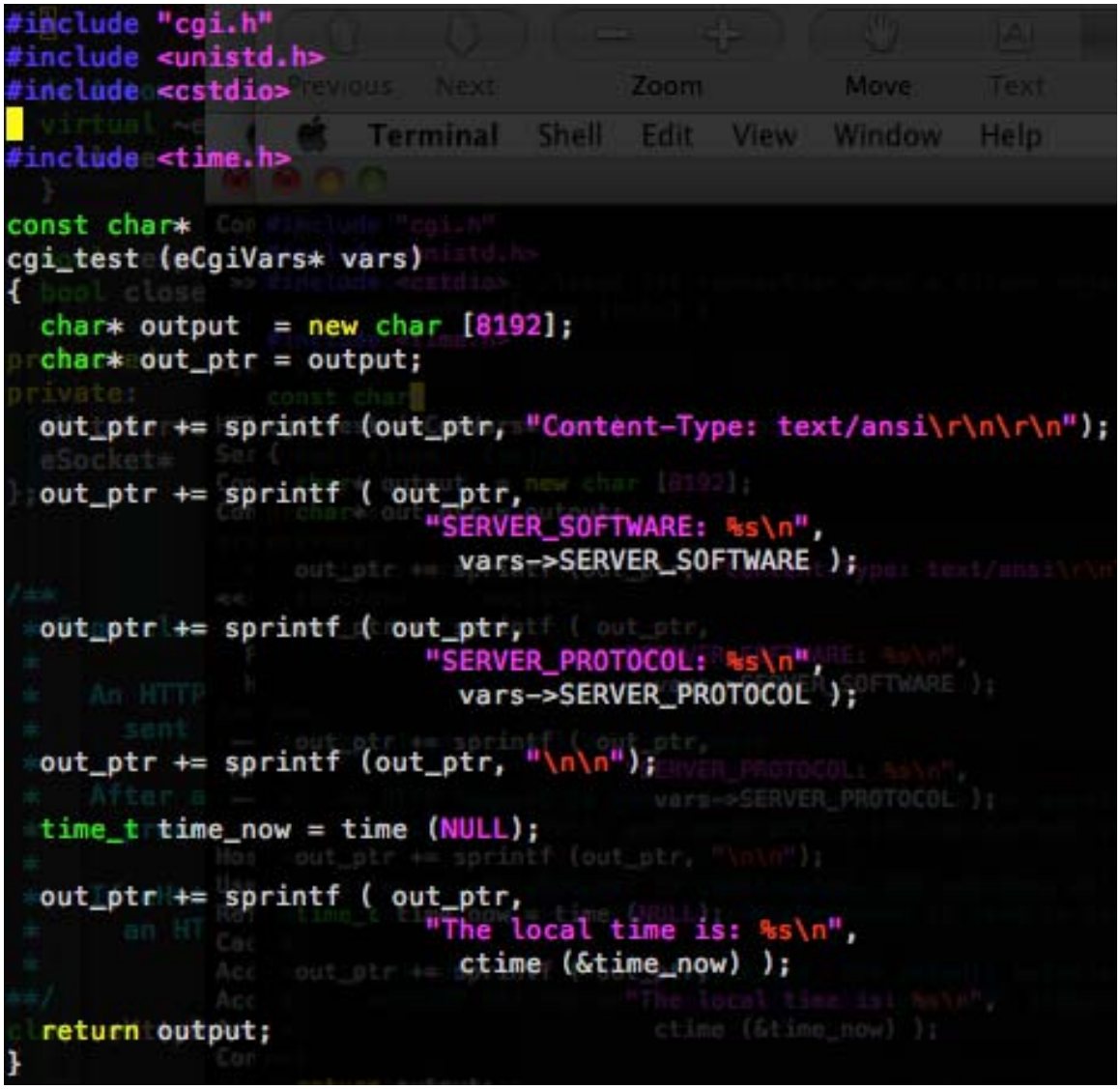
eHttpVariables contains all of the variables needed to run a CGI/1.1 program.

Depending on the mode of operation, an eCgiServer may pass a pointer to an eHttpVariables object when executing a CGI program (VFS mode – *4.2.2*), or it may export each CGI variable as an environment variable before forking to execute a CGI process (OS mode – *4.2.1*).



```
#include "cgi.h"
#include <unistd.h>
#include <cstdio>
#include <time.h>

int main (void)
{
  printf ("Content-Type: text/ansi\r\n\r\n");

  printf ("SERVER_SOFTWARE: %s\n",
          getenv ("SERVER_SOFTWARE"));
  printf ("SERVER_PROTOCOL: %s\n",
          getenv ("SERVER_PROTOCOL"));

  printf ("\n\n");

  time_t time_now = time (NULL);
  printf ("The local time is: %s\n", ctime (&time_now));

  return 0;
}
```

Figure 4.15 Sample CGI Program Source Code (OS FS)



```
#include "cgi.h"
#include <unistd.h>
#include <cstdio>
#include <time.h>

const char*
cgi_test (eCgiVars* vars)
{
  char* output  = new char [8192];
  char* out_ptr = output;

  out_ptr += sprintf (out_ptr, "Content-Type: text/ansi\r\n\r\n");

  out_ptr += sprintf ( out_ptr,
                       "SERVER_SOFTWARE: %s\n",
                         vars->SERVER_SOFTWARE );

  out_ptr += sprintf ( out_ptr,
                       "SERVER_PROTOCOL: %s\n",
                         vars->SERVER_PROTOCOL );

  out_ptr += sprintf (out_ptr, "\n\n");

  time_t time_now = time (NULL);

  out_ptr += sprintf ( out_ptr,
                       "The local time is: %s\n",
                         ctime (&time_now) );

  return output;
}
```

Figure 4.16 Sample CGI Program Source Code (VFS)

Figure 4.16 shows a sample of the source code required to implement the same CGI program seen in Figure 4.15 using a VFS. The program is invoked by binding the cgi_test (…) method to an eCallbackFile's execute callback. eVFile implements an exec (…) function that makes executing a CGI file completely transparent – the underlying file type (eDiskFile or eCallbackFile) does not need to be known at run-time.

The interface has changed slightly since the figure was created, and the function in Figure 4.16 now requires a void parameter instead of eCgiVars*. Functionally, it remains identical, however.*

4.5 Utility

4.5.1 URL Demangler

eHttpURLString is necessary to map HTTP URLs to file paths, because the HTTP protocol replaces many special ASCII characters with inline hex-codes.

For instance, the HTTP protocol sends URLs containing spaces as: “This%20URL%20Contains%20Spaces.html” – eHttpURLString demangles the string into “This URL Contains Spaces.html”, so that it maps to an actual file.

5. Conclusion

Epsilon HTTP accomplishes all that it set out to accomplish; it implements a tiny subset of the HTTP/1.1 and CGI/1.1 protocol on any system capable of compiling C++ code, with an IPv4 network stack. Furthermore, it does not require a traditional file system or an Operating System with multi-tasking capabilities to function.

While the project is technically complete, room for improvement still exists. First, during the initial development phase, it became clear that not all low-powered embedded devices use C/C++. Surprisingly, many cell phones are using high-level languages like C# and Java exclusively these days.

Second, it may be useful to mix and match virtual and OS-managed file systems in the same instance of eHTTP. The current implementation either maps paths directly to an OS filesystem, or Epsilon's proprietary VFS, but never both. In many HTTP server configurations, paths are mapped using the concept of a "virtual host", where the file system used to serve the request is derived from the hostname supplied in an HTTP request. This functionality is currently unsupported, because it was not considered important during initial requirement specification.

Third, the Virtual File System implemented is not thread-safe in its current implementation. As long as each resource mapped to a VFS is referenced by a single thread, and eVFile pointers acquired from the VFS are not kept long-term, it is possible to use the VFS-based HTTP server safely in a multi-threaded environment. However, if any of these conditions are violated, read/write behavior will be unpredictable. This is because eVFile currently stores the seek position per-file – in traditional filesystem interfaces, the file pointer is tied to a "handle" to a file, rather than the file itself. Properly correcting this will require a more sophisticated VFS design, in the mean-time it is suggested that any software written to use eHTTP using a VFS always call eVFile::seek (…) to set the position before calling read/write.

Last, the current implementation only runs on UNIX derivatives. It has been tested on Mac OS X, GNU/Linux, FreeBSD and VxWorks; it relies on BSD sockets and POSIX file APIs, which are common on traditional embedded platforms. Cell phones are among a class of embedded devices that provide the basic BSD socket and POSIX file functionality, but through proprietary APIs. Classes such as eDiskFile and eSocket will require minor refactoring to port the server to additional platforms.

A logical next step in development is, therefore, to attempt a cell phone port. This will require writing multiple code-paths across multiple languages, learning proprietary APIs and implementing sophisticated version control practices. In short, extending this project to support more platforms has even more potential educational value (with respect to software engineering) than the initial implementation.

6. References

1. J. Sirois, *VxWorks Real-Time Kernel Connectivity: Cumulative Report*, FGCU, http://itech.fgcu.edu/faculty/zalewski/CNT4104/Projects/Joanne_report_final.pdf, May, 2009.

2. http://www.windriver.com/products/vxworks/
3. http://www.microsoft.com/iis/
4. http://www.apache.org/

5. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, The Internet Society, http://www.ietf.org/rfc/rfc2616.txt, June 1999.

6. D. Robinson and K. Coar, *The Common Gateway Interface (CGI) Version 1.1*, The Internet Society, http://www.ietf.org/rfc/rfc3875, October 2004.

Appendix A. User's Manual

A1. Structure of the Distributed Files

A1.1 Documentation *Software Documentation*

- Report.doc *General design overview of the software*
- Manual.doc *How to use the software*

- eHTTP/ *API Documentation*
  - HTML/
    - index.html *Doxygen main page*
  - Doxyfile *Doxygen configuration for eHTTP*

A1.2 Source *Source Code*

- eHTTP/ *Source code for the Epsilon HTTP server*

  - Makefile *A cross-platform Makefile to compile various programs*

  - cgi-bin/ *Source code and compiled path for CGI programs*

    - cgi_test.cpp Source code for a simple CGI program (standalone)
    - cgi_test.inl Source code for a simple CGI program (inline code, for use with VFS)

  - socket.cpp/.h *Implements all classes beginning with eSocket…*
  - listening_socket.* *Implements TCP Listening Sockets (derived from eSocket).*

- http.cpp/.h — *Implements all classes beginning with eHttp…*
-
- file.cpp/.h — *Defines eVFile, eFilePermissions, eFileMode, eFileStat, and implements eDiskFile.*
- vfs.cpp/.h — *Implements eVFS, eVDirectory, eMemoryFile and eCallbackFile.*

- mime.h — *MIME Types – reserved for future use.*
- scoped_string.h — *Defines eScopedString and eScopedStringConst (Automatically frees string buffers when they go out of scope).*

- httpd.cpp — *Sample implementation of eHTTP.*
- vfs_test.cpp — *Test suite for eVFS*

A2. Installation Instructions

A2.1 Installation Overview

Epsilon HTTP's primary design goal is to provide an HTTP server that can be integrated into any C++ program. Although a standalone daemon can be compiled, it is generally only for testing platform ports. The bulk of a proper installation of Epsilon HTTP involves integrating the code for eHTTP into an existing project.

A2.2 Compiler Setup

Epsilon HTTP has been developed with a single compiler in mind, gcc. It contains a gmake compatible Makefile tailored specifically for gcc. On a UNIX platform with gcc and gmake installed, Epsilon HTTP can be compiled from the commandline using the Makefile rules described in section *A2.4*.

When integrating the eHTTP source code into an existing software project, it is suggested that a separate sub-directory be created for eHTTP. This is because some of the filenames (e.g. file.h) may collide with system-wide, or files local to a project.

eHTTP is designed to be included inline in software projects, it does not create a shared object and does not require complicated linker setup.

A2.3 Basic API Overview

In many cases, the standalone daemon for eHTTP (httpd), is insufficient to integrate Epsilon HTTP into a software project. There may be platform limitations that prevent the execution of more than one process, or the server may need to be stopped and started frequently to meet the needs of the deployed software.

Instead, the preferred approach is to completely encapsulate eHTTP within the project that uses it. This can be done with as few as four API calls…

The process for instantiating eHTTP from within a C++ program is very simple:

1. Construct an eHttpServer object

    Each instance of eHttpServer requires a dedicated TCP port, interface address, and a filesystem root.

    The constructor has three paramters:

    1. The unique port to listen on

        On many systems, ports 0-1024 require super-user privileges, so the logical choice of port 80 is often incorrect.

    2. The network address to listen on

        This address takes the form of a string.

        In the default implementation, it represents an IPv4 address or fully-qualified host name.

        A special wildcard "" is also supported, which will cause the software to listen to incoming connections on the specified port using ALL network addresses available.*

    3. Whether or not to use a Virtual Filesystem

        Recall that Epsilon HTTP has two modes of

operation with respect to File I/O:

A. Operating System Managed FS
B. Epsilon Managed Virtual FS

When this paramater is true, it causes Epsilon HTTP to construct a Virtual Filesystem.

2. Initialize the Server

Before the server can begin processing HTTP requests, it must know how to map the filenames contained in URLs to local resources.

To do this, it uses the concept of a “Root Path”.

This path can be absolute (i.e. “/home/acoleman/”) or relative (i.e. “./subdir/”), but must always be terminated by a “/”.

NOTE: Passing anything to eHttpServer::init (…) when the server was constructed using a Virtual Filesystem will have undefined behavior.

(VFS always uses “/” as its root).

3. Periodically allocate time for the Server to “think”

The server software is designed so that it does not have to “hijack” a calling thread to work. What this means, is that the software can be allocated small periods of time (defined in terms of milliseconds), with which to detect, process and dispatch I/O requests.

A software application may opt to dedicate a small slice of time within its main loop to handle HTTP requests. While the software is performing other tasks, I/O requests will pool into a buffer, which the HTTP server will respond to the next time the software calls eHttpServer::think (…).

In this way, Epsilon HTTP is capable of hosting one or more HTTP servers in a single-threaded environment. eHttpServer::think (…) takes one parameter, which indicates how much time the server is allowed to dedicate to client / server communication.

It is important to note that, the timeslice allocated to think (…) affects the establishment of connections only.

The file I/O operations required to complete HTTP requests will cause the software to block until the operation completes.

Thus, for systems that serve very large files, run complicated CGI programs, or have limited bandwidth, a dedicated thread may be desirable.

Epsilon HTTP works fine with these configurations as well, eHttpServer::think (…) can be called in an endless loop with no negative consequences.

4. Shutdown the Server

For as long as a server is running, it reserves exclusive access to its listening port. Shutting down a server releases this port back to the system.

A server is implicitly shutdown whenever its object is destroyed (i.e. goes out of scope). Manually shutting down a server is not necessary, but can be used to suspend a server without destroying it. A shut down server can be resumed by calling init (…).

If a server is shutdown while there are pending connections, the TCP port it was listening on may be unusable for a lengthy period of time. This has to do with the way Operating Systems allocate TCP ports, and there is nothing that can be done about this – think (…) should always be called before shutting down a server to avoid this problem.

The steps above are all that are necessary to start and stop an HTTP server when the Operating System Managed Filesystem mode is used. When the Virtual Filesystem is selected, additional setup is necessary.

After constructing an eHttpServer with the VFS flag set to true, and calling init (…), it maintains a unique Virtual Filesystem internally. This filesystem initially contains one directory ("/"), and nothing else.

To setup the VFS, the following additional steps are necessary:

1. Acquire a pointer to the VFS

   eHttpServer::getVFS (…)

2. Construct one or more Virtual Files, using eMemoryFile or eCallbackFile.

A. A Memory File is a named mapping to a buffer of memory within the system. Because memory can be read-only, the constructor takes an optional eFilePermission parameter.

B. A Callback File is a special file that implements the basic file I/O operations (such as read, write, stat) and advanced file I/O (such as exec) using installable callback functions.

The current implementation is limited to execution callbacks only, though the interface for read/write/stat is well defined in vfs.h.

(See "Source/eHTTP/vfs.h" for more details)

3. Add files to the VFS

eMemoryFile and eCallbackFile both inherit eVFile, which provides the file's name.

This file name contains no directory information.

Call eVFS::addFile (<directory>, <eVFile pointer>)

This function will take care of creating each directory contained within the directory string. It is not necessary to create a directory before adding a file to the VFS.

A2.4 Additional Testing Setup

A2.4.1 Makefile Rules

All of the steps discussed in section *A2.3* are implemented in Source/eHTTP/httpd.cpp. Using the supplied Makefile, httpd and vfs_test can be compiled multiple ways.

Makefile rule "embedded": (Compiles httpd in Embedded / VFS Mode)

A simple implementation of a Virtual Filesystem, where "/foobar" is a Memory Mapped File, and "/cgi-bin/cgi_test" executes the code in "Source/eHTTP/cgi-bin/cgi_test.inl".

Default Makefile rule: (Compiles httpd using an OS Manged Filesystem)

This rule compiles httpd using the Operating System for all Filesystem operations. At runtime, the server resolves files by using an optional root path parameter. If unspecified, it defaults to '.' (the current working directory).

Makefile rule "vfs_test": (Compiles a VFS Test Suite)

This rule will compile a special program called vfs_test, whose source is based on "Source/eHTTP/vfs_test.cpp", to test VFS features. The intention of this program is to teach students how the VFS works, and to facilitate debugging and future development of the VFS independent of the HTTP server.

Makefile rule "cgi": (Compiles cgi-bin/cgi_test.cpp)

The OS-based filesystem implementation of httpd requires pre-compiled CGI programs to function. The default Makefile rule ***does not*** compile any CGI programs… this rule is required to use cgi_test.

A2.4.2 Makefile Variables

The variable PLATFORM_DEFS is used to identify the server's OS in the HTTP server response strings. It may be possible to use this information to compromise system security, so a responsible HTTP server implementation must provide the option of hiding this information. Epsilon HTTP is designed so that commenting these lines out will completely remove OS information from client / server communication.

A2.5 VFS Thread Safety

Portions of the VFS are not thread safe. You can use the VFS in a multi-threaded environment, but you should NEVER acquire two or more handles to the same file in different threads. For a lengthy discussion on the topic, see "Source/eHTTP/vfs_test.cpp".

A2.6 Doxygen Documentation

"Documentation/eHTTP/html/index.html" contains detailed API information in a graphical format. This documentation was generated using a program called Doxygen, available from http://www.doxygen.org. The rules used to create the documentation are in a file: "Documentation/eHTTP/Doxyfile", and generation of the call graphs requires additional third-party software called Graphviz, available from http://www.graphviz.org.

A3. VxWorks Specific Instructions

A3.1 Introduction

A3.2 Installation

A3.3 Toolset

A3.4 Debugging

A3.5 Deployment

A3.6 Terminology

Appendix B. Source Code

The source code for this project is available from: http://satnet.fgcu.edu/~acoleman/CNT4104/eHTTP-05-01-2011.zip.

This structure of this archive is described in *Appendix A*, section *A1*. It contains the C++ source code, Doxygen API documentation, this report, and a standalone copy of the User's Manual in Appendix A.